3. Developing an analysis model

3.1 Object catalogue

3.1.1 Eskimo (Eskimo)

The Eskimo is one of the two character types in the game. He has one ability, and that is to build igloos on the field he is standing on. Each character stands on exactly one field at a time. Characters can lose life points, fall into water and be pulled out by other characters. Characters can move under their own power, i.e. they can move to a neighbour of a field they know. They also need to be able to use objects from the ones they have in their bag.

3.1.2 Arctic Explorer (Explorer)

It is similar to the Eskimo, except that it cannot build igloos, but it can search for corners.

3.1.3 Stable Field (StableField)

When characters are moving, fields keep track of which character is where, and provide the concept of adjacency, i.e. they know and can tell which other fields they are adjacent to. In addition, they can be manipulated by objects, such as digging up objects, and snowstorms can generate snow on them.

3.1.4 Unstable Field (DangerousField)

They differ from a stable field in that they have a capacity, so if they have more than one player on them, they will cover them all in water.

3.1.5 Glove (Glove)

He can shovel snow from a field, dig up an object, and put the acquired object in the bag of the character using it.
This object is always present in every character, and is different from other objects.
All other items, this one too, are placed in the character's bag and can be used (i.e. ordered to be used) by the character.

3.1.6 Food (Food)

When used, it will erase itself from the using character's bag, in exchange for increasing the number of life points by one.

3.1.7 Shovel (Shovel)

It's very similar to a hand-held needle, you can dig snow with it, and you can dig objects with it. The difference is that it's a completely generic object, you can get it just like the others.

3.1.8 Wetsuit (DivingGear)

It's a relatively passive object, all it does is that if they see it in a player's bag, it won't die if they fall in the water.

3.1.9 Rope (Rope)

It is the responsibility of the user to move the saved character to the user's field.

3.1.10 Bag (Inventory)

Each character has a bag, in which they store all their belongings. The bag's job is to keep track of them and to make them accessible to the character through it.

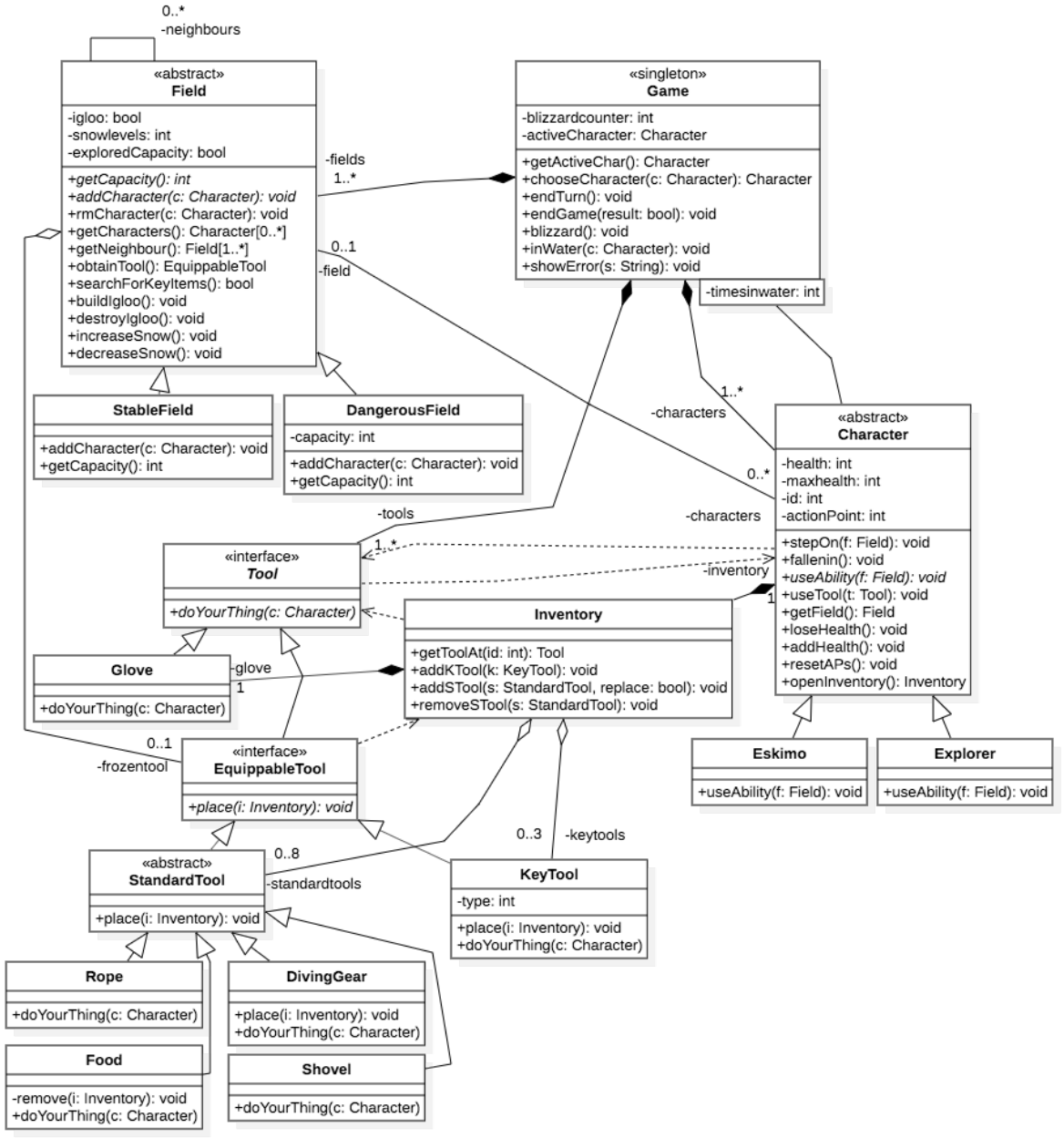
3.1.11 KeyTool (KeyTool)

The job of the key items is to signal the end of the game when they are used according to the conditions given.

3.1.12 Game (Game)

Perhaps the most complex object in terms of liabilities. Tasks include managing a blizzard, killing characters and announcing the end of the game, dealing with characters who have fallen into the water, and managing circuits.

3.1 Static structure diagrams



3.2 Description of classes

3.2.1. Character

- **Responsibility**
  This abstract class represents the characters in the game. It knows and manages your state, e.g. how much life you have, how many actions you can still take, what your ID is, etc. Characters know the Field they are on and have an Inventory. All this can be queried.
- **Ancestral classes: -**
- **Interfaces: -**

- **Attributes:**
  health [int]
  This attribute indicates how much life the character has at a given moment.
  id [int]
  This is the unique identifier of the character to reach it. Its value is constant and is assigned at the start of the game.
  actionPoint [int]
  This value determines how many more actions the character can take in the current turn. At the beginning of each turn, it takes the value 4, and then decreases by 1 for each action.
  game [Game]
  Direct knowledge of the Game singleton makes it much easier to communicate messages such as death or waterfall.
  inventory [Inventory]
  This attribute manages and stores the assets that the character can use. Each character has exactly one inventory, i.e. it can never be NullPtr.
  field [Field]
  This attribute specifies the field the character is currently in.

- **Methods**

  void stepOn(f: Field)
  Moves the character to the field received as a parameter space, i.e. changes the attribute of the field, and then signals the field to accept another character.

  void fallenIn()
  If the field you would move to can no longer hold more players, you can use this method to tell the Game that the player is in water.

  void useAbility(f: Field)
  Abstract method that is responsible in descendants for ensuring that a character of a given type uses the appropriate ability. (In our case, the Eskimo builds igloos and the polar explorer measures capacity.) In the end, it uses an action point.

  void useTool(t: Tool)
  On the Tool interface object received as a parameter, calls the method to use it, and uses an action point.

  Field getField() This
  method returns the reference of the field the character is currently in.

  void loseHealth()
  This method is called to remove a life from a character. If his life is thus reduced to 0, it is reported to the Game.

void addHealth()
This method gives the player extra health in case it does not exceed the maximum allowed.

void resetAPs()
After the player's turn, the number of action points available is expected to decrease somewhat. To get it back to 4 in the next round, this method must be called.

Inventory openInventory()
This method returns a reference to the Inventory of the given player.

3.2.2. DangerousField

- **Responsibility**

  This class represents the dangerous fields that make up the ice field, i.e. fields from which it is possible to fall into the water.

- **Ancestral Classes**
  Field -> DangerousField
- **Interfaces: -**
- **Attributes**

  capacity [int]
  This constant attribute specifies how many characters the field can hold.

- **Methods**

  int getCapacity()
  This method returns the capacity of the given field

  void addCharacter(c: Character)
  This method places the character in the field, given as a parameter. It checks if the capacity has been exceeded, if so, it signals to the Game class that the characters on it have been flooded.

3.2.3. DivingGear

- **Responsibility A**
  standard device that has a passive ability, if the character has it, it can survive in the event of a waterfall without being pulled out. It has a fixed place in the toolbox at location 1, it will put itself there when it is placed in the toolbox.
- **Ancestral classes -**
- **Interface:** StandardTool
- **Attributes: -**
- **Methods:**
  void doYourThing(Character c):
  It does nothing, it has no active ability.

void place(Inventory i):
It puts you in position 1 in the toolbar.

3.2.4. EquippableTool

- **Responsibility**
  (Acquirable tool) An interface representing a tool that the character can acquire by digging out of the fields and place in his own toolbox.
- **Ancestor classes:** tool -> EquippableTool
- **Methods:**

  void place(Inventory i)
  Abstract function. Places the asset itself in the player's inventory, with respect to where that asset group might be placed.

3.2.5. Eskimo

- **Responsibility**
  A descendant of the Character class, he represents the Eskimos in the game. His ability is to build igloos, which he can only do on his own field if he has snow on it. His maximum life is 5.
- **Ancestral classes:** character -> Eskimo
- **Interfaces: -**
- **Methods:**

  void useAbility(f: Field):
  This method builds or destroys the igloo on the field it is currently on by calling the appropriate method on it. The igloo is not necessarily built, as it can only be built on a snowy field, and destruction is always successful. It also deducts one action point at the end.

3.2.6. Explorer

- **Responsibility:**
  a descendant of the Character class, he represents the polar explorers in the game. His ability is exploration, which he can only perform on his own or adjacent fields. His maximum life is 4.
- **Ancestor classes:** character -> explorer
- **Interfaces: -**
- **Methods:**

  void useAbility(f: Field):
  This method determines the capacity of the field given as a parameter. It is placed in an action point. If a non-adjacent or own field is passed, you cannot determine it, but no action point is deducted.

3.2.7. Field

- **Responsibility:**
  this abstract class represents the ice sheets that make up the ice field. It knows and manages its own state, e.g. whether it has snow on it, whether it has buried tools on it, whether it has igloos built on it, etc. Fields know their neighbouring Fields, they can be queried.
- **Ancestral classes: -**
- **Interfaces: -**
- **Attributes:**

  igloo [bool]
  This attribute indicates whether an igloo is currently present in the given field (true if present, false if not). The value can be changed by building or destroying igloos.

  snowlevels [int]
  The snowlevels attribute specifies how many layers of snow are currently on the field, this value will never be greater than 5. The value may change with snow shoveling, igloo building, or the arrival of a storm.

  exploredCapacity [bool]
  During the game, a polar explorer can determine the capacity of the field (whether it is stable, if not, how many people it can hold). This attribute indicates whether it has been done. Once it becomes true, it remains true until the end of the game.

  neighbours [Field[0..*]]
  This array contains the fields adjacent to the field, depending on the shape and location of the field, it can have as many neighbours as you like (obviously for practical reasons you shouldn't have more than 6). This attribute is constant throughout the game, it contains the same fields.

  characters [Character[0..*]]
  This array contains the characters currently in the field. A new character is added when someone steps on it, or when someone is pulled onto the field from the water, a character is removed when it steps off, or is pulled onto another field.

  frozentool [Tool]
  This attribute specifies the buried tool (if there is a tool buried in the given field)

- **Methods**

  int getCapacity()
  Abstract method that returns the capacity of a given field. It returns different values for a stable and a dangerous field, hence the need for abstraction.

  void addCharacter(c: Character)
  Abstract method that places the character received as a parameter in the field. Abstraction is needed because in case of a dangerous field, you need to check if the capacity is exceeded, but in case of a stable field, everything is fine.

void rmCharacter(c: Character):
If a character escapes from a given field, this function is used to signal it to the field. The avoiding character is passed as a parameter.

Character[] getCharacters():
This method can be used to retrieve the array/collection/list of characters currently in the given field

Field[] getNeighbour():
this method can be used to retrieve the array/collection/list of fields adjacent to a given field.

void obtainTool():
this method resolves to dig out a tool frozen in a field, and returns a result if the field does not contain a tool. If an EquippableTool is frozen in the field, it is dug out and placed in the character's toolbar.

bool searchForKeyItems():
This method searches the Inventory of the characters in the given field. If it finds that they have all three components, it returns true, otherwise false. In this case, all characters must be in the same field, so it checks that too.

void changeIgloo():
calling this method builds or destroys the igloo on the field, i.e. the igloo attribute becomes false if it existed before and true if it did not exist before. The method increases the snow cover on the field to 1 when it is destroyed and decreases it to 0 when it is built.

void increaseSnow():
this method increases the snow layer in the field by 1 if it has not reached 5, and leaves it as it is if it has 5.

void decreaseSnow():
This method decreases the snow layer in the field by 1 if there is still a snow layer in the field.

3.2.8. Food

- **Responsibility A**
  standard item that the character can consume, regenerating 1 body heat (but not exceeding the maximum). Once the food is consumed, the item disappears from the toolbar.
- **Ancestral classes -**
- **Interface:** StandardTool
- **Attributes -**
- **Methods**

void doYourThing(Character c):
Removes a layer of snow from the field where the player is standing.

3.2.9. Game

- **Responsibility The**
  class that implements the engine of the Model architecture. It stores the elements of the game. It handles incoming calls from the controller, calculates who is the next character, interacts with it. It manages the rounds, the turns and the events that occur at the end of the rounds (snowfall, keeping track of players in water). When rescued, it offers the user the choices that the character can make to be rescued. Starts and ends the game. There can be only one instance.
- **Ancestral classes: -**
- **Interfaces** -
- **Attributes**

  characters [Character[1..*]]:
  Stores the characters in the game.

  tools [Tool[1..*]]:
  Store the tools in the game.

  fields [Field[1..*]]:
  Stores the fields in the game.

  activeCharacter [Character]
  Reference to the next character (of the current characters), can be interacted with

  blizzardCounter [int]
  After a snowfall (within given limits), a randomly generated counter indicates how many rounds before snowfall is expected again

  timesinwater [Character, int]:
  keeps a count of the characters that are currently in water in a map, and how long they have been in water.

- **Methods**

  Character getActiveChar()
  Returns a reference to the character that is currently in the queue

  void endTurn()
  Handles end of turn events: sets up a new active character, gives him 4 working points; checks if the next player is not still in the water without a wetsuit, otherwise calls*endGame()*. If the *blizzardCounter* expires, i.e. snowfall is due, calls *blizzard()*.

  void blizzard():
  Increases the snow layer by 1 for a random number of fields between given bounds. Subtracts a body heat from characters that are not in igloo.

void endGame(bool result):
the function that handles the end of the game, telling users that the game is over and with what result.

void showError(String s):
Some interactions are only allowed under certain conditions, if they are not present, the action cannot be performed and the user will be notified via Game.

Character inWater(Character c):
is told that character *c* is in water, so it updates *timesinwater*.

Character chooseCharacter(Character c):
the work done by R*ope* results in this call. In this case, the *Game* will list people who have fallen into the water in character field *c* (circle) and offer a choice to the user, who selects which character to pull out of the water. The character that is pulled out is removed *from timesinwater.*

3.2.10. Glove

- **Responsibility A**
  tool that is available to each character by default. It allows a character to remove a layer of snow from the field where he is standing. It has a fixed place in the *Inventory*.
- **Ancestral classes -**
- **Interface:** tool
- **Attributes -**
- **Methods**

  void doYourThing(Character c):
  Removes a layer of snow from the field where the player is standing.

3.2.11. Inventory

- **Responsibility:**
  this department is responsible for storing, distinguishing and managing assets. It classifies, separates and manages the assets. Since in reality there can't be an infinite number of devices per person, Inventory can store up to 12 devices at a time.
- **Ancestral classes: -**
- **Interfaces: -**
- **Attributes**

  glove [Glove]
  This attribute represents the glove that players have with them from the start of the game. The location of the Glove is at index 0.

standardtools [StandardTool[0..8]]
This container contains the tools that can be picked up, but are not key, such as rope, food, etc. They can be accessed with index 1..9, of which if a wetsuit is received it will come to index 1 anyway.

keytools [KeyTool[0..3]]
This container contains the pickup and key tools such as gun, beacon, cartridge. They can be accessed with index 10..12

- **Methods**
  Tool getToolAt(id: int)
  Finds the tool with the appropriate index and returns its reference

  void addKTool(k: KeyTool)
  Places a key tool in an unoccupied index location, given as a parameter

  void addStool(s: StandardTool, replace: bool):
  Places a non-key tool given as a parameter in an unoccupied index location. The replace indicates whether the tool has a higher priority (typically for wetsuits), in which case it will be inserted anyway.

  void removeSTool(s: StandardTool):
  Removes the non-key tool received as a parameter, e.g. if food is eaten, it logically ceases to exist in one's inventory. From this point on, the tool will be able to occupy its place again.

3.2.12.KeyTool

- **Responsibility**
  Key item to acquire. Only 3, but different items (FlareGun, Cartridge, Flare) can be found in the whole game. Their ability is to assemble and fire, so they start searching for key items on the same field, if all 3 are found on 1 field, they fire the flare gun, the game ends in victory.
  They are stored in a separate place in the toolbox.
- **Ancient Classes -**
- **Interface:** EquippableTool
- **Attributes:**
  type [int]
  tells you which of the 3 objects you can find is the one you are looking for
- **Methods**

  void doYourThing(Character c):
  they start searching for the key items on the same field, if they find all 3 on 1 field, they fire the flare gun, the game ends with a win. In this case, all the characters must be on the same field, check that too.

void place(Inventory i):
Placed in the character's toolbar, in the space reserved for key items.

3.2.13.Rope

- **Responsibility A**
  standard means by which a character can pull a waterlogged companion from an adjacent field. If more than one character is waterlogged and can be rescued, the player can choose.
- **Ancestral classes -**
- **Interface:** StandardTool
- **Attributes -**
- **Methods**

  void doYourThing(Character c):
  strikes a character in an adjacent field if possible, if there are more than one, the user chooses one of the offered ones. This is done by calling the *chooseCharacter()* method on the Game, which returns the character to be pulled out, and pulls it out to its own field.

3.2.14. Shovel

- **Responsibility A**
  standard tool that allows a character to remove 2 layers of snow from the field he is on at the same time, for one working point.
- **Ancestral classes -**
- **Interface:** StandardTool
- **Attributes -**
- **Methods**

  void doYourThing(Character c):
  Removes 2 layers of snow from the field where the player is standing (at most if possible).

3.2.15.StableField

- **Responsibility**
  This class represents the stable fields that make up the ice field, i.e. fields from which it is not possible to fall into the water.
- **Ancestral Classes**
  Field -> StableField
- **Interfaces: -**
- **Methods**

int getCapacity()
This method returns -1, indicating that it has no capacity

void addCharacter(c: Character)
This method places the character given as a parameter in the field, i.e., it puts it into the characters array

3.2.16.StandardTool

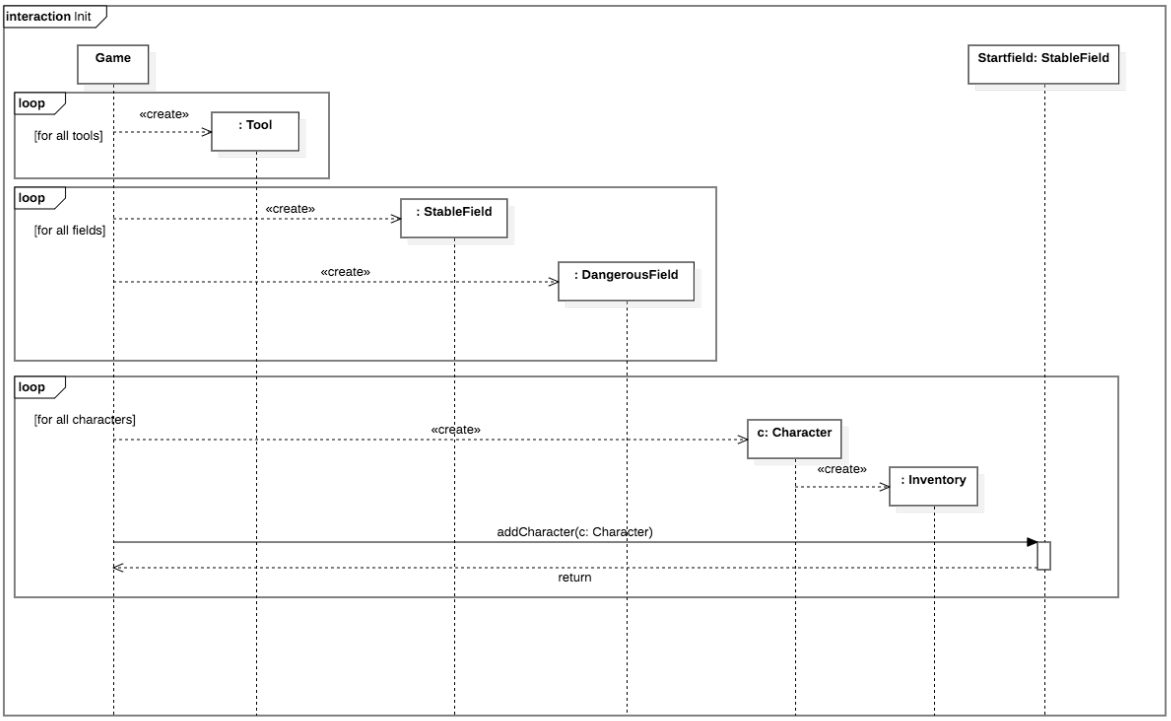
- **Responsibility**
  Abstract class representing an obtainable asset that is not an integral part of the game, i.e. not a key object. When a new tool is added to the game, it is placed here. They have a defined place in the toolbox. A character can have up to 8 of these items.
- **Ancestor classes:**
  tool -> EquippableTool -> StandardTool
- **Methods**
  void place(Inventory i)
  Places the tool itself in the player's inventory, which can only be placed in the [2-9) position in the inventory

3.2.17.Tool

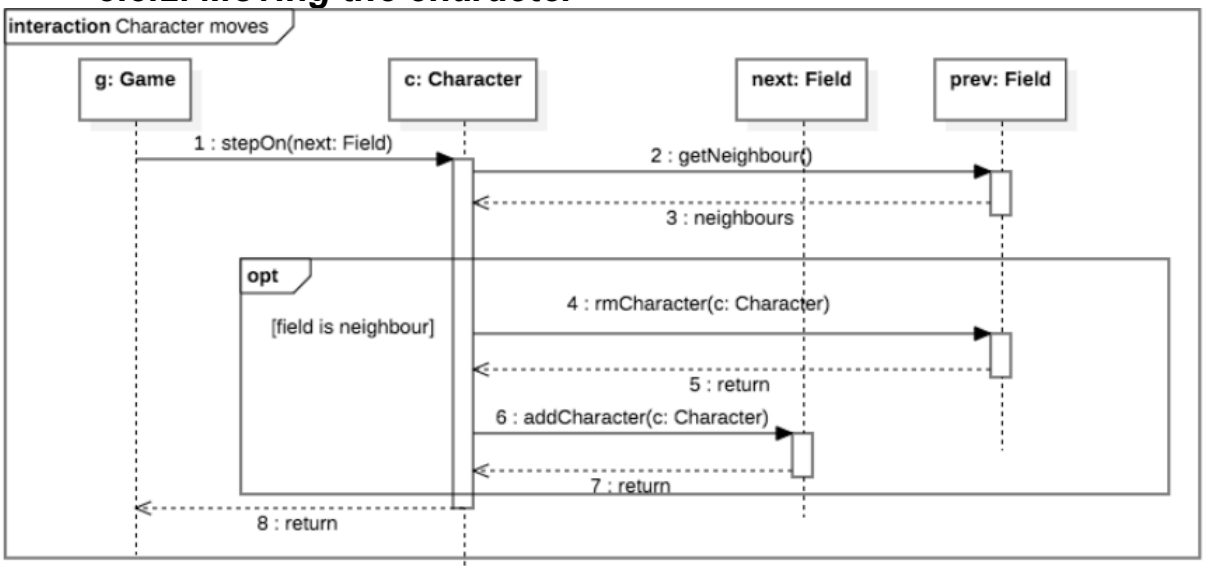
- **Responsibility**
  Interface representing devices. Each device has its own ability that the character (who owns it) can use. They cost 1 working point to use.
- **Ancestral classes -**
- **Methods**
  void doYourThing(Character c):
  (Abstract method.) The tool will do its job if the conditions are right.

3.3 Sequence diagrams

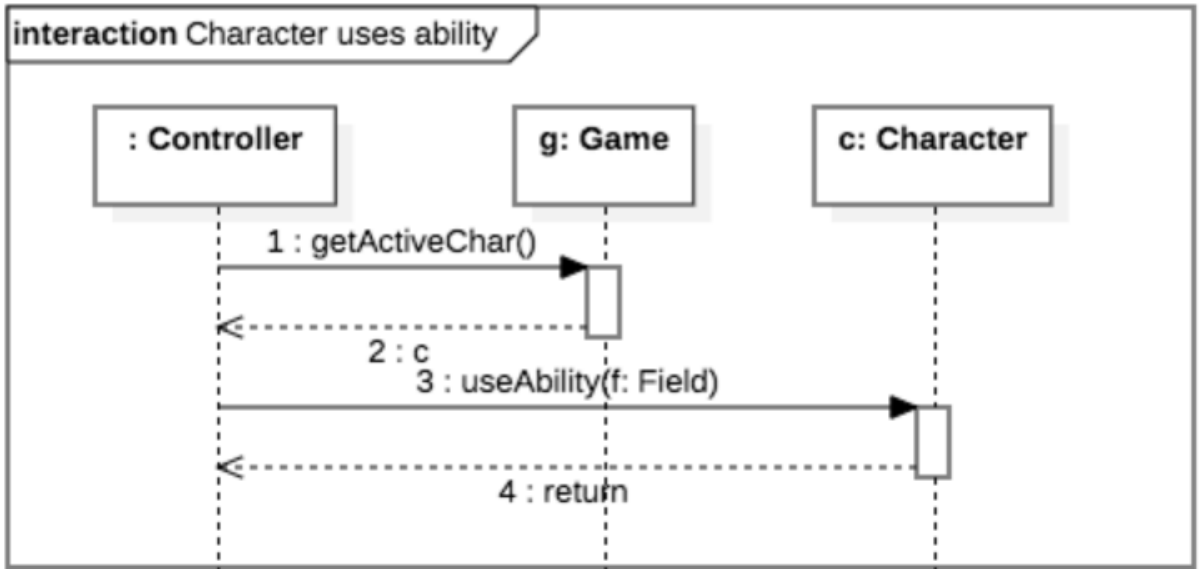
3.3.1. Creating objects



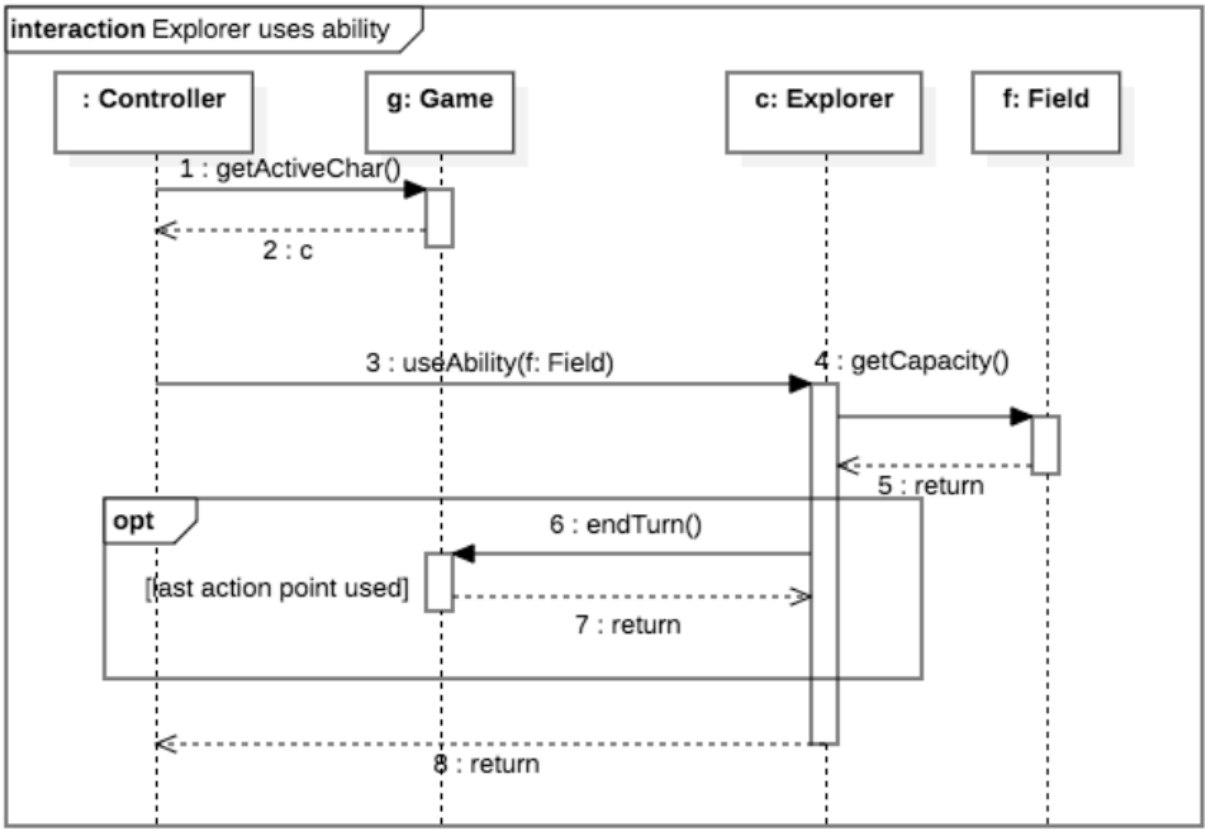
3.3.2. Moving the character



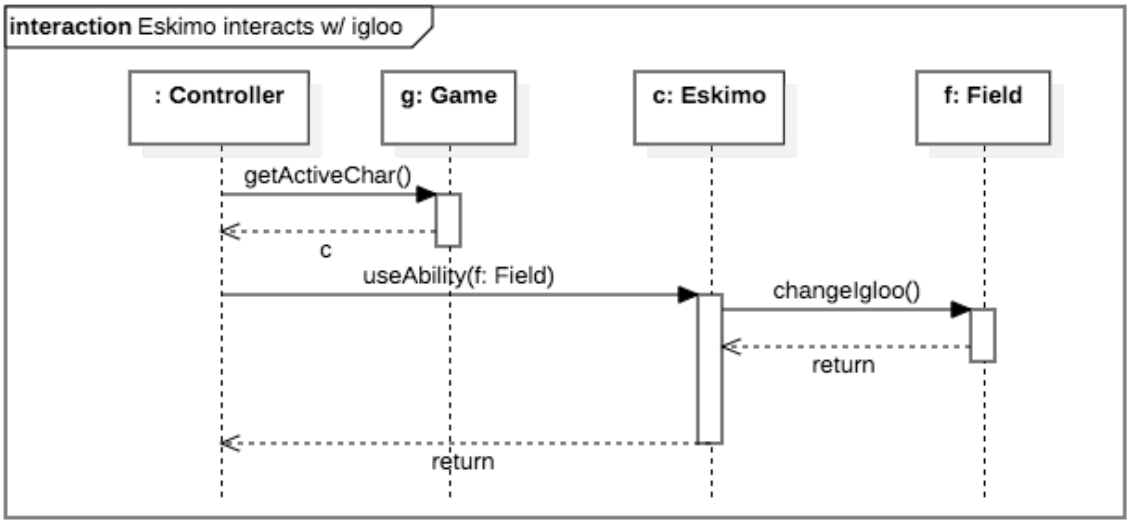
3.3.3. Character uses his/her ability



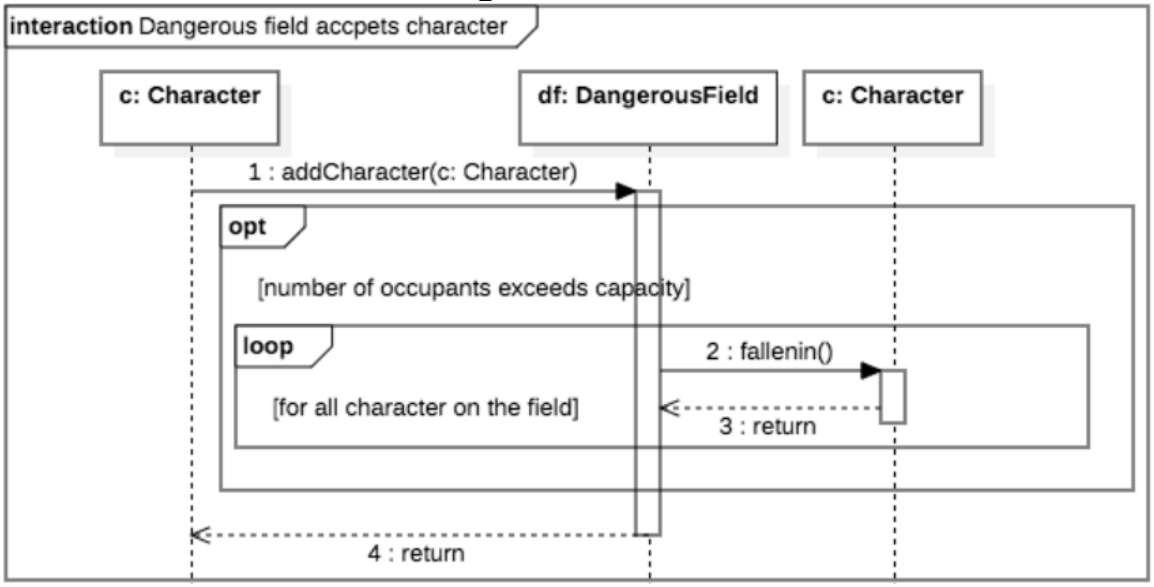
3.3.4. Polar explorer checks the load-bearing capacity of the ice sheet



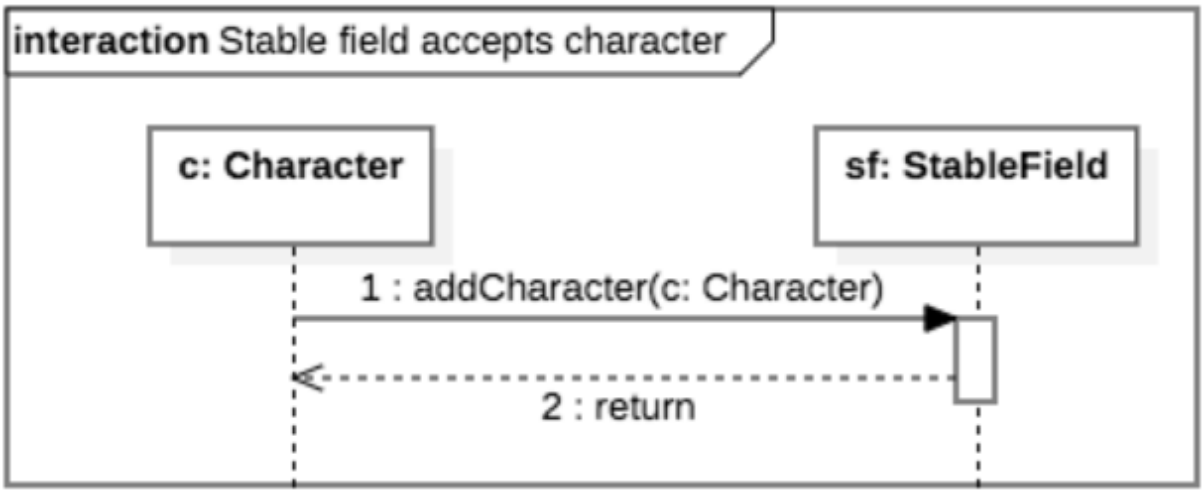
3.3.5. Build/build Eskimo igloos



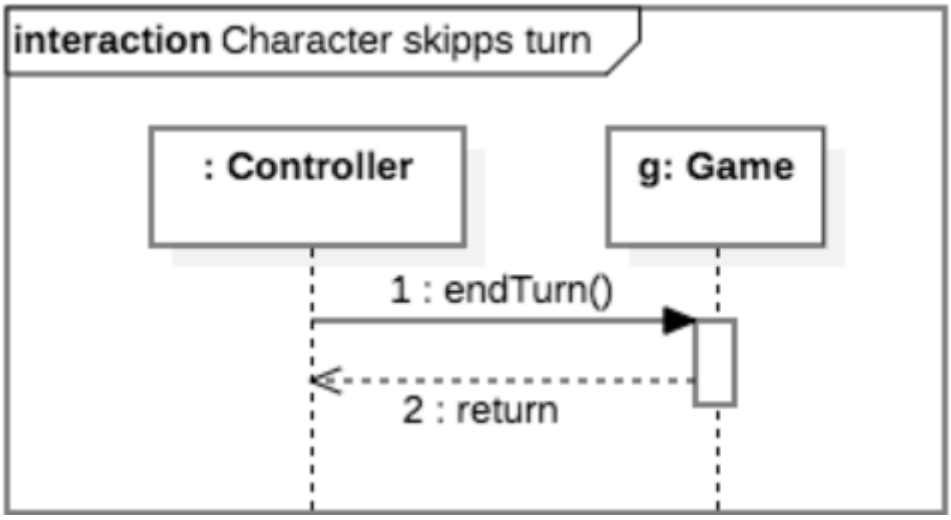
3.3.6. Character enters dangerous field



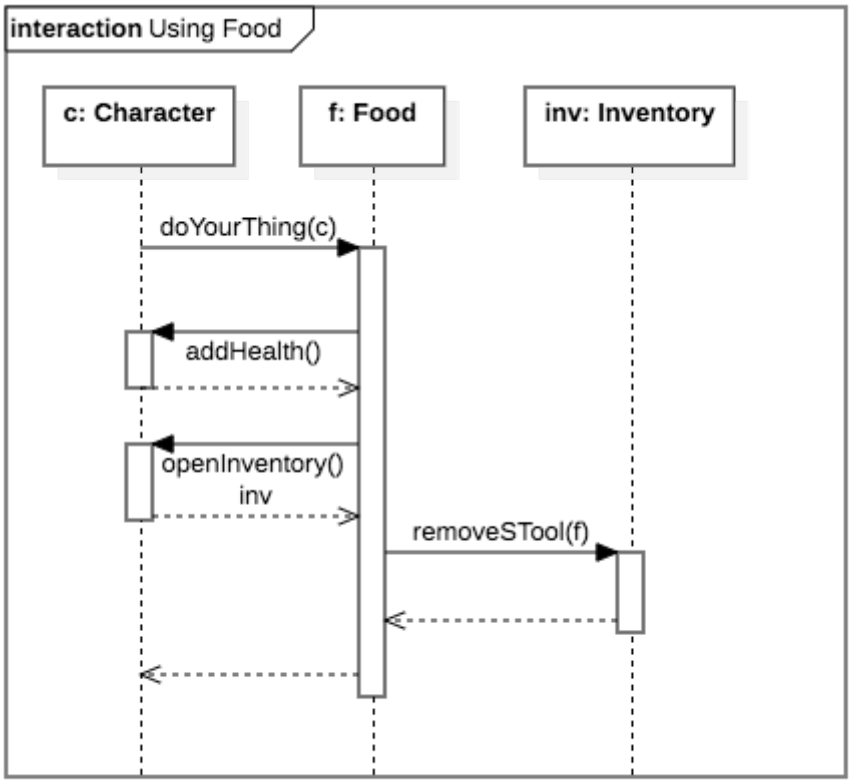
3.3.7. Character enters stable field



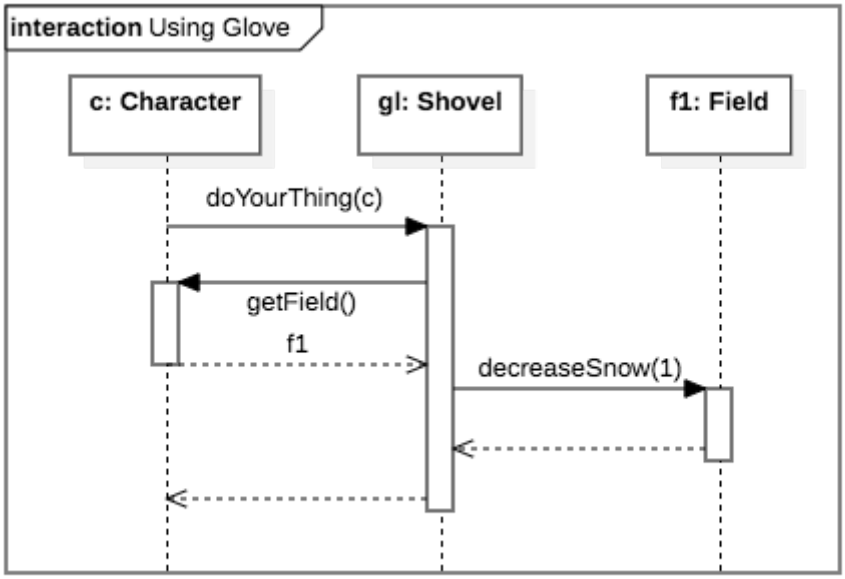
3.3.8. Skipping a round



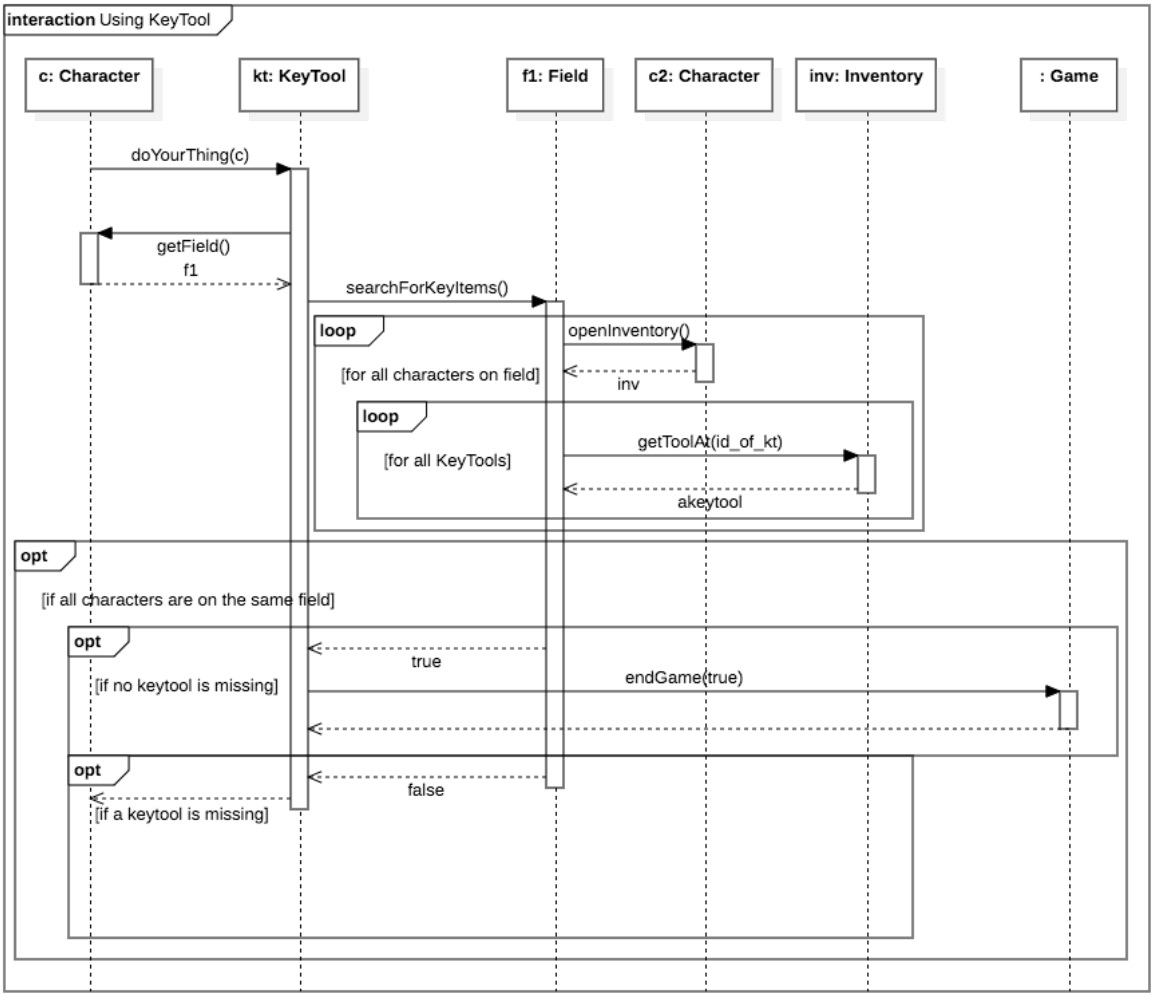
3.3.9. Use of food



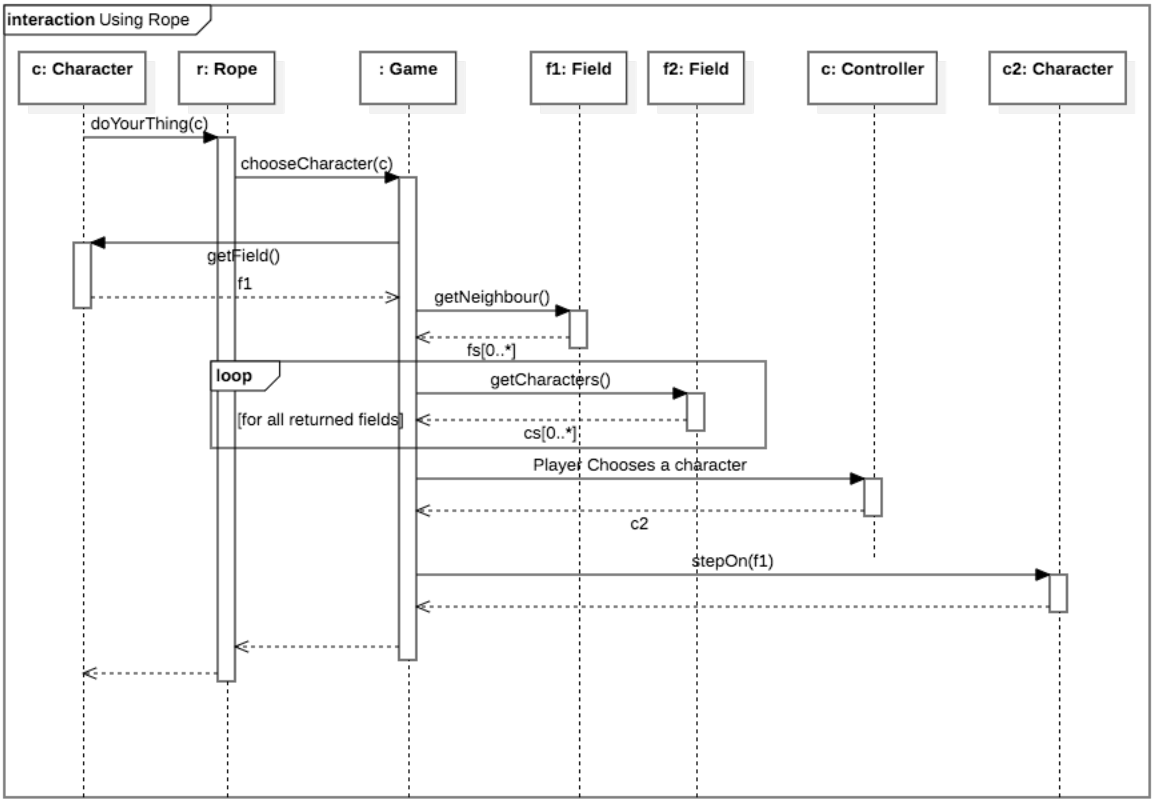
3.3.10. Use of gloves



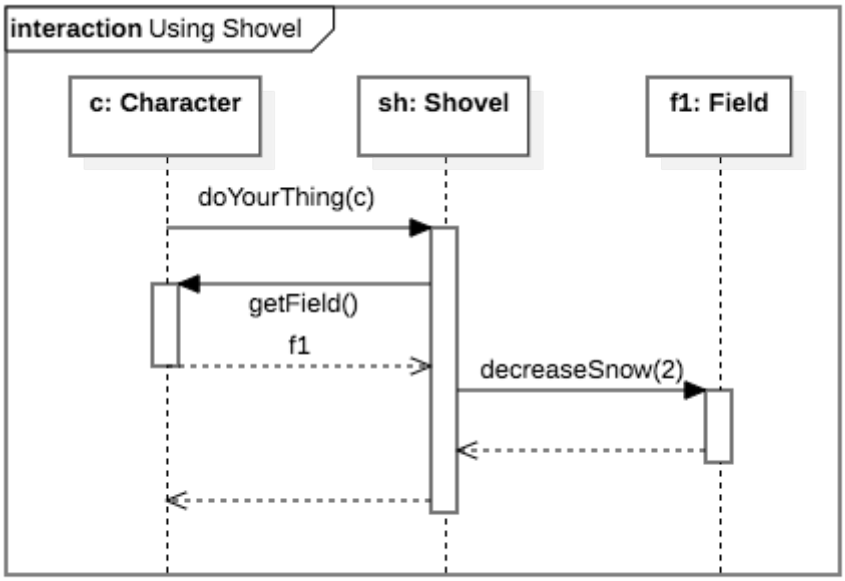
3.3.11. Use of a key fob



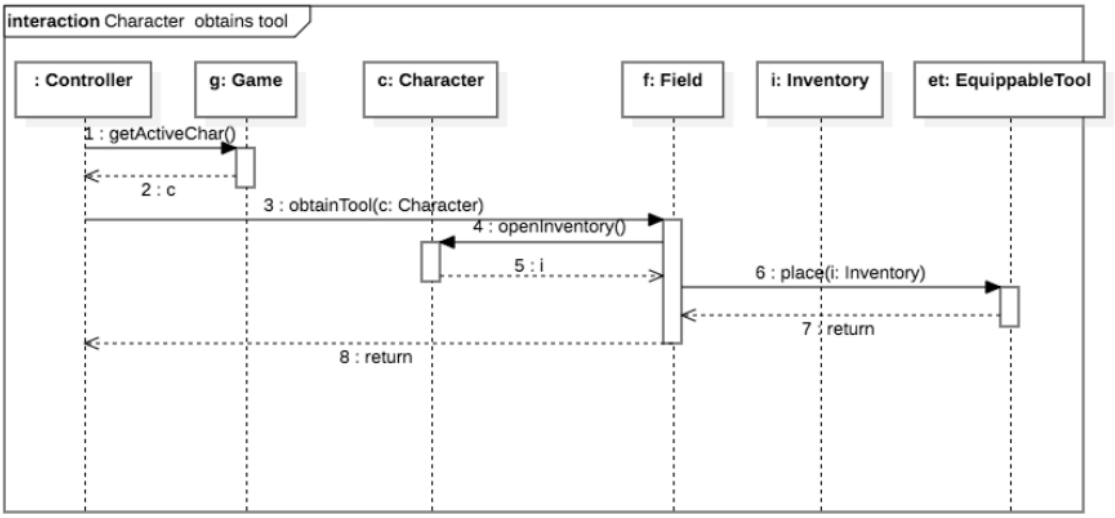
3.3.12. Character rescue



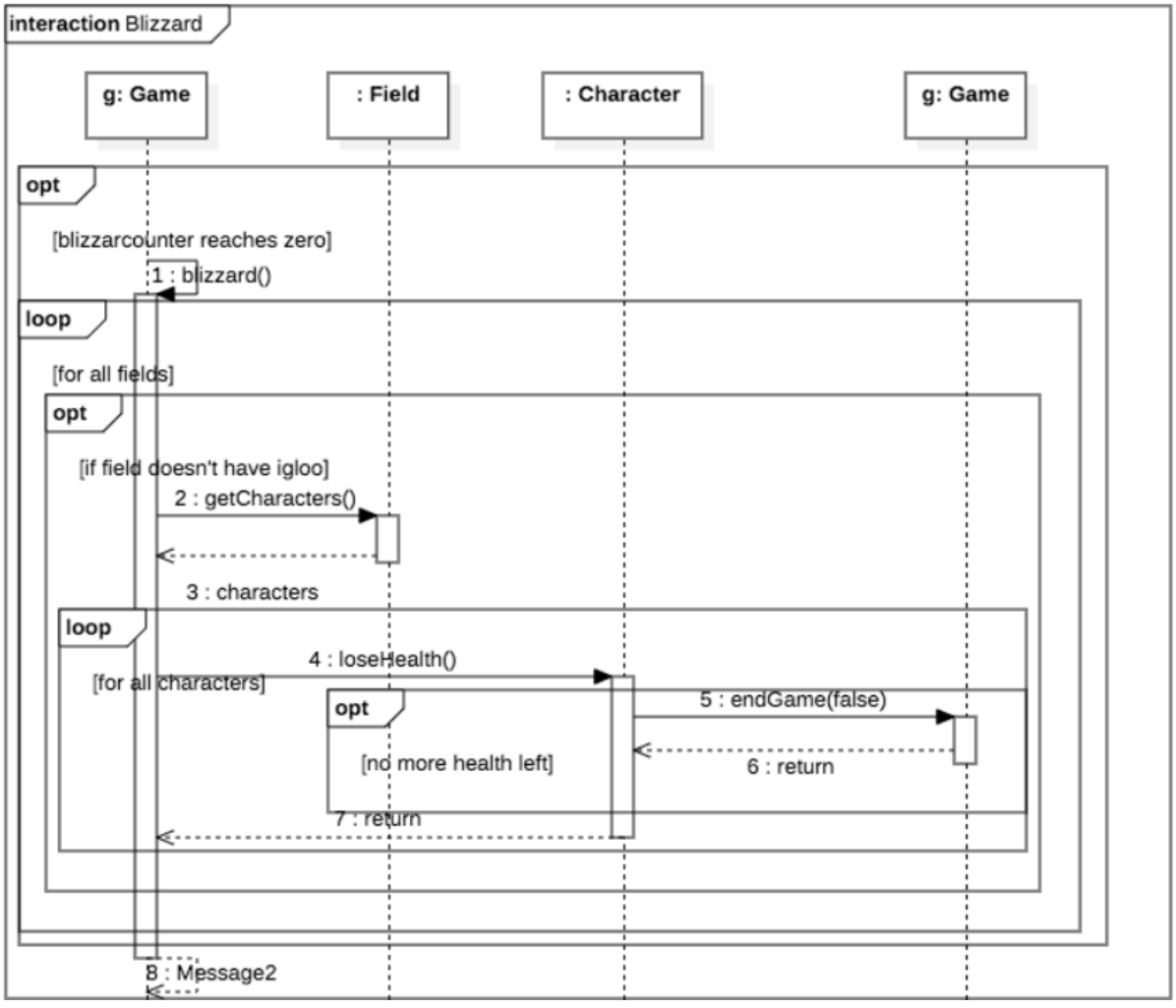
3.3.13. Snow shovelling



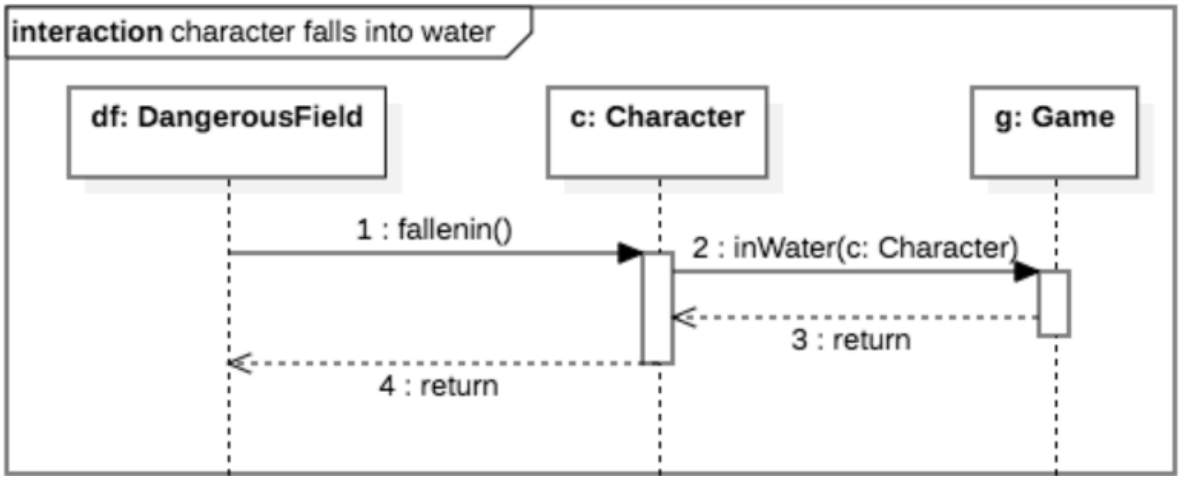
3.3.14. Excavation of the object



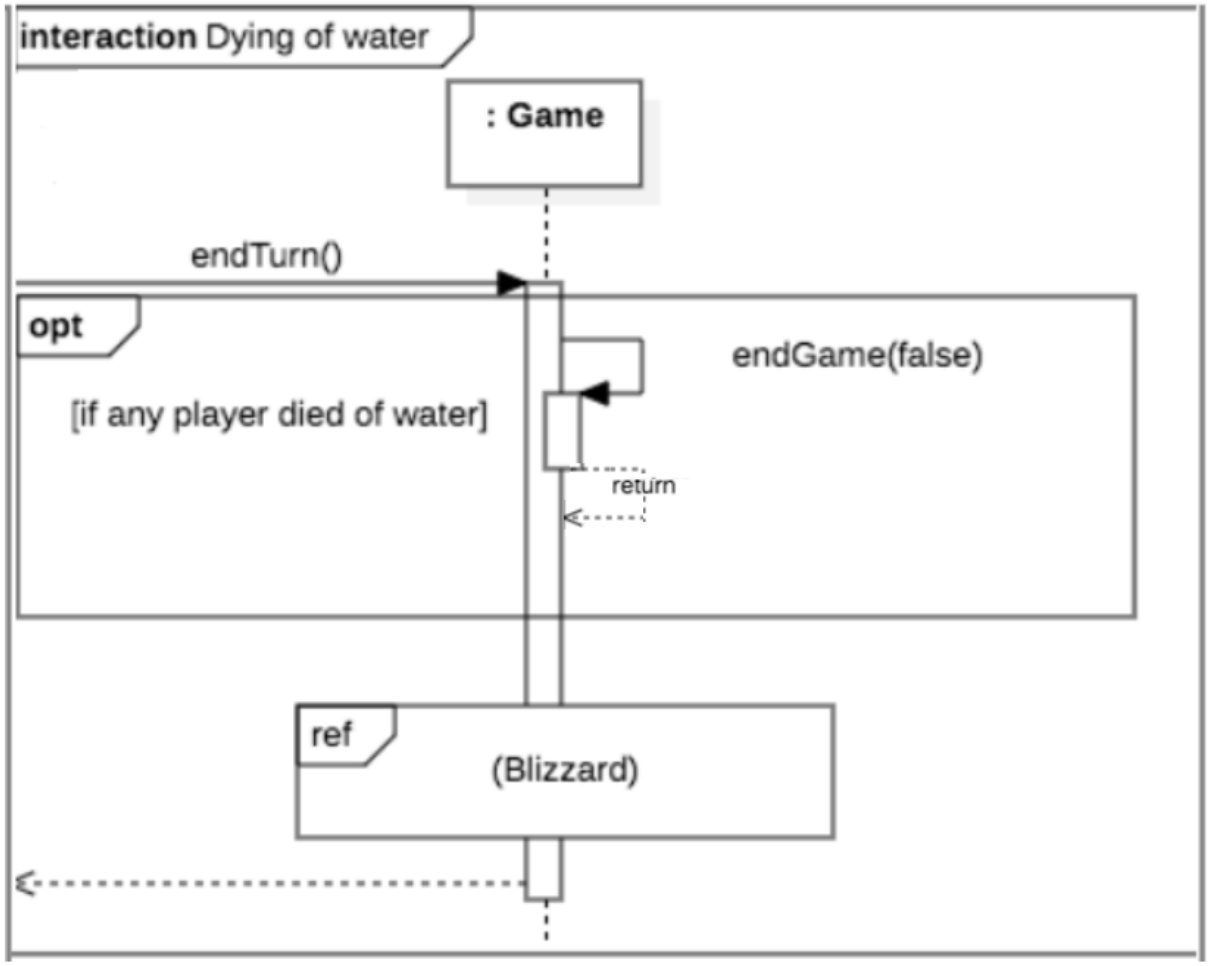
3.3.16. Storm



3.3.16. Character falls into water



3.3.17. End of round exercises



3.4 State-chartok

We did not use states or state machines to solve this problem.

3.5 Diary

Start at	Duration	Participants	Description
2020.02.24. 12:00	2 hours	Digging Banker Retrieved from Juhász Lukács	Face-to-face meeting Topic: assignment of tasks
2020.02.24. 7:00	1 hour	Digging	Object catalogue
2020.02.25 14:00	4 hours	Hain Juhász	Developing a class diagram outline
2020.02.25 16:00	1 hour	Digging	Object catalogue
2020.02.28 20:00	2 hours	Retrieved from	Detailed elaboration of a given part of the class diagram (tools) + preparation of descriptions
2020.02.29. 12:00	2 hours	Juhász	Detailed elaboration of a given part of the class diagram (character, field) + descriptions
2020.02.29.	3 hours	Banker	Preparing sequence diagrams
2020.02.29.	3 hours	Lukács	Sequence diagrams
2020.02.29 20:00	1 hour	Digging Banker Retrieved from Juhász Lukács	Team meeting on Discord, check progress, review completed tasks
2020.03.01.	1 hour	Juhász	Formatting a document
2020.03.01.	1,5	Banker	Inspection, repair